Designing efficient and safe weak references in Eiffel with parametric types

Frederic Merizen, Olivier Zendra, Dominique Colnet

Designing efficient and safe weak references in Eiffel with parametric types

Frederic Merizen* Olivier Zendra†

Dominique Colnet‡

Emails: {merizen,zendra,colnet}@loria.fr

LORIA
615 Rue du Jardin Botanique
BP 101
54602 VILLERS-LES-NANCY Cedex
FRANCE

December 15, 2003

Abstract

In this paper, we present our work on the design and implementation of weak references within the SmartEiffel project. We introduce the known concept of weak references, reminding how this peculiar kind of references can be used to optimize and fine-tune the memory behavior of programs, thus potentially speeding up their execution. We show that genericity (parametric types in Eiffel) is the key to implementing weak references in a statically-checked hence safer and more efficient way. We compare our solution for weak references to similar notions in other languages and stress the advantages it offers.

Keywords: Eiffel, SmartEiffel, genericity, parametric types, safety, weak references, memory management, optimization

1 Introduction

Weak references allow a program to maintain an object reference that does not prevent the garbage collector from considering the object as dead and from reclaiming the associated memory. There are many practical uses for weak references: caching data that is expensive to compute, recycling objects to lessen memory pressure on the garbage collector, keeping meta information about data without maintaining the data itself alive, and implementing observer/observable relationships.

*Henri-Poincaré University, Nancy 1 / LORIA.
†INRIA-Lorraine / LORIA.
‡University of Nancy 2 / LORIA.

Normally, when the application code references an object, the latter is considered accessible, belongs to the live set and may not be collected by the garbage collector. This is a "normal", *strong reference* to the object. Weak references on the contrary allow the application to access the object, but do not prevent the garbage collector from collecting it. How come this does not lead to dangling references ?

Actually, to access a weakly referenced object, the application must obtain a strong reference to the object. If the application obtains this strong reference before the garbage collector runs, then the garbage collector may not collect the object because a strong reference to the object exists. But if the application asks for the strong reference after the object has been collected, a `Void` value[1] will be returned instead. By testing for this `Void` value, the application can know whether the object has been collected and act consequently.

In this paper, we aim at providing weak references for a high-level language, Eiffel, so that they can be easily compiled to code that is both efficient and safe. We show how *generic types* — Eiffel *parametric types* — are a great asset to reach these goals. Indeed, although similar concepts exist in various languages, our solution is the only one we know of that is based on generic types, which as we will show makes it safer and more efficient. The implementation issues we faced are explained within the context of SmartEiffel, The GNU Eiffel Compiler, which we develop at LORIA and that compiles Eiffel source code to portable C code or Java bytecode.

Note that we do not consider in this paper special kinds of weak-reference-like objects such as ephemerons which are peculiar to solutions for the finalization problem.

This paper is organized as follows. First, section 2 presents a brief survey of existing concepts similar to weak references in various languages. Section 3 then introduces our solution for safe weak references in Eiffel, based on parametric types. Section 4 discusses and explains our design and implementation choices and how our solution generalizes. Finally, section 5 concludes and opens perspectives for future work.

2 Weak-reference-like concepts in various languages

2.1 Proxy objects

In the weak reference concept we presented in the introduction, weakly referenced objects are accessed by obtaining a strong reference to them, and then working with this strong reference like with any other normal reference.

The Python language implements another, seemingly less cumbersome mechanism called *proxy objects*. Such proxies weakly reference an object, while exposing the very same interface as the object itself. All calls to the proxy's features are relayed to the proxied object if it still exists, otherwise an exception is thrown.

Python proxies are a syntactic sugar that hides one level of indirection, and is functionally equivalent to weak references. However, they are somewhat slower since the liveness of the referenced object has to be checked *each time* it is accessed. More importantly, they encourage a poor programming style by

[1] `Void` is the Eiffel equivalent of `null` in C, C++, Java...

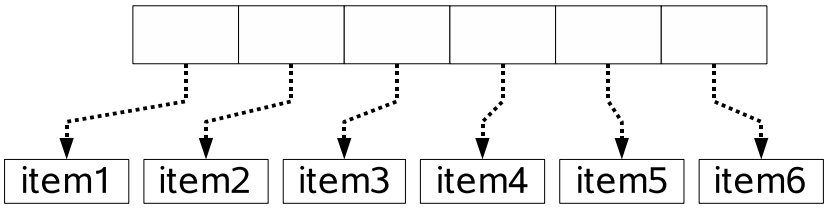


Figure 1: A weak vector

letting the developer use proxies (almost) anywhere normal objects are expected. Therefore, the introduction of a proxy in one single place can lead to unexpected exceptions being thrown in any part of the program, which makes correctness proving or even simply debugging a daunting task. Note that interestingly, Python also provides plain weak references.

2.2 Reference strengths

Weak references were made available in Java [GJS96] in the 1.2 specification through the `java.lang.ref.Reference` class and its descendants. Each Java `Reference` can be added to a queue that is used for asynchronous finalization. More interestingly, Java offers three flavors of weak references, corresponding to three different "strengths". They can be sorted in order of decreasing strength:

Soft references instruct the garbage collector to *try and keep the referenced object alive* for some time after it ceases to be strongly reachable. Yet, objects that are only softly reachable may be destroyed, especially if they have been in that state for some time or memory becomes scarce. This kind of reference is well-suited to implement caches.

Weak references *do not keep objects alive*: if an object is found to be weakly reachable only, it is to be destroyed by the garbage collector.

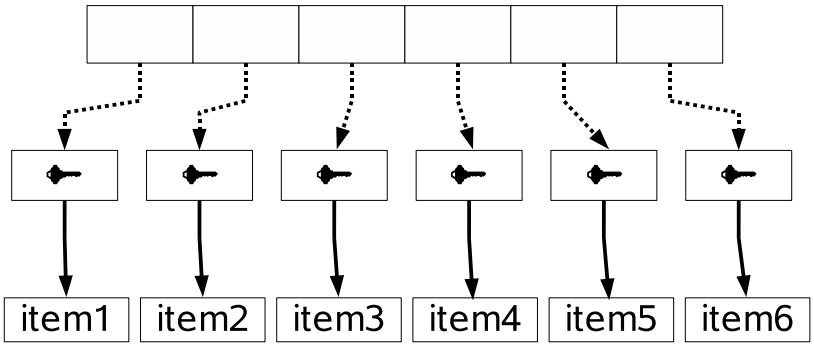
Phantom references are finalization tools, and cannot be dereferenced. They share the queue mechanism with the other two types of weak references.

2.3 Weak collections

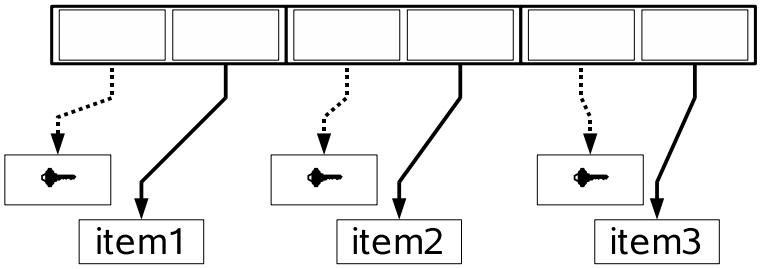
Weak references are typically used for groups of objects rather than isolated objects. Therefore, many languages, including Java and many Smalltalk [GR83] dialects offer one or more of the following weak collections.

Weak vectors are similar to vectors or arrays, except for the fact that their elements may be destroyed by the garbage collector (see figure 1).

Weak key dictionaries hold key-value pairs. The key is weakly referenced, and when a key is collected by the garbage collector, the whole pair is discarded from the dictionary. This works as if the value were strongly referenced by the key — see figure 2(a). However, it is not possible to actually add a reference inside the key, which is why the dictionary itself holds a strong reference to the value in addition to the weak reference to



(a) Concept



(b) Implementation

Figure 2: Weak key dictionaries. Plain lines represent strong references, dotted line weak references.

the key, and an extra finalization mechanism to discard the value when the key goes away — see figure 2(b).

Weak value dictionaries also hold key-value pairs, but they weakly reference the value. Of course, when a value is collected, the corresponding pair is discarded from the dictionary.

Doubly weak dictionaries hold key-value pairs that weakly reference both the key and the value. If either is collected, the pair is discarded from the dictionary.

In some languages including Guile[2], weak collections are the only kind of weak reference available.

Weak vectors are ideal to implement object recycling, and weak dictionaries can be used to attach meta information to objects without keeping the objects live more than necessary.

[2]Guile [Fre] is the GNU implementation of the Scheme language.

3 Designing safe weak references with parametric types

The previous section shows that the concept of weak reference is not new and has been integrated in several languages. However we think those weak references do not offer the best safety and performance possible.

In the current section, we introduce our solution to better design and implement weak references. We first consider the conceptual model, then the application developer point of view, and finally the compiler and garbage collector implementation aspects, in an Eiffel compiler.

3.1 A conceptual model for weak references in Eiffel

In the model we designed for weak references in Eiffel, the garbage collector always frees the memory used by an object when it discovers this object is not strongly referenced anymore. Our notion of weak reference is thus similar to Java's. The user shall then obtain a `Void` reference informing her/him the object has been collected and is no longer available through the weak reference.

We think this is the most general and convenient kind of weak reference. Indeed, it does not alter the referenced object lifetime, which is highly desirable or even mandatory to semi-automatically manage memory and to instrument the garbage collector. Furthermore, this kind of weak reference makes it possible to realize interesting data caches, if garbage collection is not too frequent. Finally, we consider this kind of reference as a sound basis upon which improvements such as soft references or ephemerons could be built if required.

Note that Eiffel normally features two kinds of objects: "normal" ones, which are (strongly-) referenced objects, and so-called *expanded objects*, that is objects passed by value (see [Mey92] for details). While it obviously makes plenty of sense to *weakly* reference objects which are normally referenced, it seems to us a complete nonsense to try and weakly *reference* expanded objects that normally can only be used by value. Our model hence forbids weak references on expanded objects.

Finally, one very important aspect, though not directly related to the semantic model of our weak references, is that they should be as efficient as possible when used, and cost absolutely nothing when not used. Indeed, integrating a new possibility in a language and the corresponding compiler(s) should not negatively impact things when this concept is not used.

3.2 The user side: the `WEAK_REFERENCE` class

The interface we chose to offer weak references in Eiffel to the application developer is a very simple and convenient one: a `WEAK_REFERENCE[G]` class.

This class is used via two routines:

- `set_item`, to set the weak reference so that it weakly references an object
- `item`, to query the weak reference, in order to obtain a strong reference on the weakly referenced object, or `Void` if the latter has been reclaimed by the garbage collector

```
class WEAK_REFERENCE[G]
--
-- Weak reference to an object.
-- This kind of reference does not prevent the object from being
-- reclaimed by the garbage collector (in which case item returns Void).
-- Item makes it possible to get (a strong reference to) the object.
-- Inheriting from this class is prohibited.
--
creation
   set_item
feature
   item: G;
         -- Return a (strong) reference to the object
   set_item(i: like item) is
         -- Set the object to be weakly referenced
      do
         item := i
      ensure
         item = i
      end
end -- WEAK_REFERENCE
```

Figure 3: The `WEAK_REFERENCE` class.

As a consequence, the typical way to access a weakly referenced object involves the following steps:

1. Get a normal — strong — reference on the object by querying the weak reference object.
2. Check this strong reference is not `Void`, otherwise nothing can be done.
3. Work with the non-`Void` reference as with any normal reference.
4. Take care not to maintain the object live with the strong reference once it is not in use anymore. If the reference is a local variable, it is automatically canceled at the end of the routine, otherwise it may be wise to set it to `Void` explicitly.

The `WEAK_REFERENCE[G]` class (shown in figure 3) is very simple, since it only contains one attribute, or instance variable, named `item`, which is seen by the developer as an argumentless function[3], and one procedure, `set_item`. Except for the uniform access to the `item` attribute, this is very similar of what can be done for other languages.

However, this class features a fundamental peculiarity, allowed by the power of the Eiffel language: it is a *generic class*.

Generic classes are the Eiffel parlance for classes implementing *parametric types*. As can be seen from its name, the `WEAK_REFERENCE[G]` class accepts one type parameter, symbolized by the formal parameter name '`G`'. The application

[3]This is called the uniform reference principle of Eiffel, see [Mey92].

developer provides an actual type parameter when using the `WEAK_REFERENCE[G]` class: for example, s/he declares a variable of type `WEAK_REFERENCE[IMAGE]`, another of type `WEAK_REFERENCE[CUSTOMER]`, etc. This is much more than a syntactic shortcut: the `WEAK_REFERENCE` class may never be used alone, it does require the extra piece of information that is the type of the object it references. A `WEAK_REFERENCE[IMAGE]` is a completely different kind of object, type-wise, than a `WEAK_REFERENCE[CUSTOMER]`, as much as an `IMAGE` is different from a `CUSTOMER`. This means that even when using weak references, the program remains *fully and statically typed*.

This is a great asset for safety, since the correctness of the typing can be checked at compile time, whereas in other languages without parametric types, such as Java, a weak reference only contains an `Object`, which has to be back-cast at run-time to its actual dynamic type when the weak reference is queried. Parametric weak references also allow a better efficiency: in Java for example the cast is executed by the JVM at run-time, which incurs a cost, while our Eiffel weak references based on parametric types are compiled to the most efficient code. Indeed, each actual derivation of the `WEAK_REFERENCE[G]` class is compiled as a specific type, differently from the others. The actual type parameter thus determines the appropriate code to set and access the weakly referenced object.

As far as we know, no comparable solution for weak references based on parametric types exists, in any language.

3.3 The compiler side: impact on the garbage collector compilation

This section presents the impact of our weak reference implementation within the SmartEiffel compiler and with respect to the specialized mark-and-sweep garbage collector[4] that is automatically generated for each compiled application. Most of the workings of the compiler and the garbage collector fall beyond the scope of this paper; more details can be found respectively in [ZCC97, CZ99, ZC01] and [CCZ98].

3.3.1 Actually weakening the reference

The `WEAK_REFERENCE[G]` class as presented in figure 3 is not the only thing needed to have weak references in Eiffel. Indeed, when looking at its code for the `item` attribute, it seems it references quite normally — that is to say *strongly* — the referenced object and thus prevents it from being reclaimed by the garbage collector.

To get the correct behavior and actually weaken the reference, it is of course necessary to slightly modify the garbage collector code, which is generated by the compiler and automatically adapted to the compiled application. The modification lies in the generation of the marking functions. The function generated for the marking of `WEAK_REFERENCE`s has to be slightly different from other marking functions, since it must *not* propagate the marking process to its `item` attribute. Therefore, it just has to mark the weak reference object itself.

Note that this is an example of the power of code customization for optimization: the marking routine for `WEAK_REFERENCE[G]` objects is the simplest

[4] Section 4.3 page 11 discusses implementation issues with other kinds of garbage collectors.

and the most efficient one, done by assigning a field a value. This very simple routine is thus an ideal candidate for inlining, which further reduces its cost.

3.3.2 `Void`-ing the reference when necessary

Beside making the weak reference actually weak, a second aspect is crucial to fully respect the semantics of the model we defined above: the reference has to be nullified, or `Void`-ed in Eiffel vocabulary, when the garbage collector decides to reclaim the weakly referenced object. As explained in section 3.1, in this first simple version of weak references, this decision is made as soon as the collector knows the object is not longer strongly referenced.

Setting the `item` attribute of the `WEAK_REFERENCE[G]` to `Void` is performed quite logically during the sweep phase of garbage collection, after the mark phase has identified all the references and reachable objects in the system.

In the garbage collector that SmartEiffel generates, all objects are segregated by *type*. With memory chunks which are type-homogeneous, there is one sweeping routine for each type. This routine is specialized, in the sense that it knows were to find each object mark flag in the chunk, and statically knows the size separating two objects.

Setting the `item` to `Void` for `WEAK_REFERENCE[G]` implied modifying these sweeping functions to look at the mark flag of the object pointed by the `item` field. Indeed peeking at the mark flag of the object pointed by the `item` field makes it possible to know whether this object is still strongly referenced from somewhere else or not; in the latter case `item` is set to `Void` in the weak reference object. However, this modification is not as easy as the one described in section 3.3.1. Two issues complicated things: first, finding the weakly referenced object header, then understanding the status of this object, garbage-collection-wise.

Finding the weakly referenced object header In the memory layout of objects compiled by SmartEiffel, the object "header" that contains the mark flag lies *after the object* itself[5]. But `item` holds a pointer to the object that — like all object references in SmartEiffel — directly point to the *beginning of the object* payload, not to its internal bookkeeping header. Therefore, to access this header of the weakly referenced object, the sweeping function for the weak reference needs to know the size of the object to add it as a negative offset to the `item` pointer.

This is not an issue for objects which are instances of leaf types, that is types without heir. In this case indeed the object size, hence the location of the header, is known at compile time. The modification to the sweeping function for a kind of weak reference on leaf object thus relies on this size and generates a simple constant code to access the header.

However when a type is polymorphic, that is it has heirs, whose instances may have different sizes, the size of the object referenced by the `item` field may not be known at compile time. The size thus has to be found at run-time, according to the dynamic type of the referenced object. This dynamic type is easy to find: such polymorphic types always contain a type identifier, used to resolve polymorphic calls as described in [ZCC97]. The sweeping function

[5] Except for resizable objects (arrays), but these are expanded objects and thus not subject to weak referencing, as explained in section 3.1.

that is generated for a specific kind of weak reference on a non-leaf object thus has to be adapted to read this type identifier (which is always the first field of the object) and then access a table that associates to the type identifier the size of its instances. Note that this table is not only useful for weak references management, but also for the debugger.

Understanding the status of an object When the mark flag of an object that may be weakly referenced is found, its meaning is not immediate anymore, while it would be with normal objects, because now we are "peeking ahead" of the normal sweeping process.

The word that holds the mark flag in the weakly referenced object may also hold a pointer that is used for chaining free memory slots. Thus three states are possible for this word:

1. It may contain a pointer
2. It may contain `FSOH_MARKED`
3. It may contain `FSOH_UNMARKED`

The first case — a pointer — means the object has been found to be a dead object and has just been added in the free list of slots for this type[6]. In this case, the weak reference has to have its `item` field set to `Void`.

The second case — a `FSOH_MARKED` flag — means the object has been marked as live (strongly referenced) by the mark phase. The weak reference thus remains unchanged.

The third case — an `FSOH_UNMARKED` flag — is trickier. Indeed, since we are sweeping a weak reference and thus peeking at the mark flag of its `item` object, the latter may already or may not yet have been swept. If it has not been swept yet, the `FSOH_UNMARKED` flag means the object has not been marked live by the mark phase and is thus not strongly referenced anymore. As a consequence, the object is bound to be added into a free list later, when its memory chunk is swept. The weak reference thus has to have its `item` field set to `Void`. On the contrary, if the weakly referenced object has already been swept, the `FSOH_UNMARKED` flag means the object had been marked live by the mark phase, and then reset by the sweep phase to `FSOH_UNMARKED` in order to prepare it for the next mark-and-sweep cycle. In such a case, the weak reference must remain unchanged.

We thus see that knowing whether an object has already been swept is crucial when weak references are used, because of our "peek-ahead" for the `item` field. A small yet greatly effective change in the code generated by SmartEiffel made this possible, at no cost. Making sure objects chunks — hence objects — were swept monotonously, that is for example by increasing addresses, is enough. The sweeping routine generated for weak references just has to compare the address of the object pointed by the `item` field to that of the weak reference currently being swept.

[6]The free lists are rebuilt from scratch at each collection, as explained in [CCZ98].

4 Discussion

In this section, we further discuss and explain some conceptual choices we made when designing our solution for safe and efficient weak references. We first address the keyword versus library component issue, then the by value versus by reference choice. Finally, we discuss how the solution we proposed in this paper generalizes, especially to other garbage collectors and languages.

4.1 Keyword versus library component

Weak references could be provided as a language extension instead of a standard library class. However, implementing a keyword inside the parser would freeze the syntax, making it harder to experiment with weak references. Furthermore, the design of Eiffel (see [Mey92]) is focused on keeping the core language relatively small to avoid unexpected interactions. Finally, if other types of versatile references (soft references, etc.) were to be added, a keyword for weak references would imply either other keywords, or a kind of sub-language to distinguish them all. A new class thus seemed less invasive, more flexible and more scalable.

4.2 Expanded or referenced weak reference

Since `WEAK_REFERENCE[G]` are Eiffel objects, they apparently could be either expanded or referenced — this boils down to a pass by value versus pass by reference choice. We decided our weak reference objects would be normal, referenced objects, not expanded ones. Let's see the reasons and implications of this choice.

4.2.1 Ease of use

Weak references are more often found in arrays than alone — just think of caches (see section 2.3 page 3). Therefore, it is crucial that arrays of weak references work seamlessly.

However, if `awr` is an array of *expanded* weak references, a call to `awr.item(i)` returns a *copy* of the weak reference found in the i^{th} position of `awr`. It follows that the natural Eiffel idiom for changing the object that is weakly referenced by a weak reference in an array, `awr.item(i).set_item(another_object)`, does not work for expanded weak references. Indeed, it makes a copy of the weak reference, and then applies `set_item` to this copy, instead of changing the reference that is stored in the array.

The correct way to change the object that is weakly referenced by a weak reference in an array is to use an auxiliary weak reference that is set to point to the wanted object and then is put into the array:

```
aux_weak_ref.set_item(another_object)
awr.put(aux_weak_ref, i)
```

This solution is quite cumbersome and counterintuitive, which makes expanded weak references very error-prone, all the more so as the solution that does not work is syntactically correct and does not trigger a compiler error or a warning.

Ease of use thus favors using normal, referenced `WEAK_REFERENCE[G]` objects, instead of expanded ones.

4.2.2 Efficiency

When compared to expanded objects, referenced objects incur a memory penalty of two machine words: one word for the pointer to the object, and another word for the object header. Weak references are tiny objects since they consist of a single pointer and only weight one machine word. So making them referenced objects implies a memory overhead of 200 %. Nonetheless, we expect weakly referenced objects to be fairly large, otherwise there would be no point in *weakly* referencing them. Therefore, the memory overhead should be negligible when taking into account the weakly referenced object.

4.2.3 Correctness

In SmartEiffel, expanded local variables are stored on the stack or in processor registers rather than in the heap. This is where the conservative part of SmartEiffel's semi-conservative garbage collector [CCZ98] kicks in, and treats every word in the stack that seems to be a pointer to an object like an actual reference, by marking the referenced object as live. Since weak references contain an `item` pointer, if one ends up in the stack, the object pointed by `item` will be marked... hence *strongly* referenced. While this misidentification may not cause any faulty program behavior, it would increase memory usage and negate the weak aspect of weak references that are put onto the stack. This would include all weak references declared as locals, or declared as attributes of expanded locals, and all weak references passed as arguments. All this seems a bit restrictive.

Of course, we could try to conceal the pointer in the weak reference from the garbage collector. For example, a pointer might be effectively disguised by XORing it with a well-chosen bit pattern. But then, we would have to decide what to do when an object gets collected while it is referenced by a weak reference from the stack. The normal thing to do would be to `Void` the `item` field of the weak reference, but this is impossible.

Indeed, since SmartEiffel's garbage collector currently has no knowledge about the types of the objects that lie in the stack — which is why it has to be semi-conservative — it is unable to decide whether a word that looks like a weak reference is actually one or not. Setting them all to `Void` would lead to critical errors, modifying words in the stack that were actually not weak references. The opposite policy also fails, because not setting to `Void` the weak references means that subsequent calls to `item` on them would return dangling pointers. Hiding weak pointers from the collector is thus not possible with a conservative algorithm.

To sum it up, expanded weak references can be implemented correctly, but at the cost of a "reduced weakness" caused by misidentifications for weak references found in the stack. On the contrary, referenced weak references are fine, since they live in the heap, where the garbage collector is non-conservative (a.k.a type accurate).

4.3 Generalizing our solution

Although we rely on a semi-conservative mark-and-sweep collector in the SmartEiffel Eiffel compiler, our solution for weak references is also applicable to other

kinds of collectors, and to other object-oriented languages — provided they feature parametric types.

In section 3.3.1 page 7, we detailed the modification to mark-and-sweep have it actually weaken the weak reference. Of course, this also applies almost verbatim to other kinds of collectors that rely on a marking phase followed by a kind of reclaiming phase, such as mark-and-compact collectors and copying ones. However, if a non-marking garbage collector were relied on, for example a reference-counting algorithm, things change a bit. There, the appropriate modification to actually weaken the reference would have consisted in changing the handling of pointer assignments — including argument passing — so that the reference counter of an object is *not* changed when this object is assigned to the `item` attribute of a weak reference, and of course not changed either when `item` is reset to `Void`.

In section 3.3.2 page 8, we addressed the issue of setting the `item` attribute of the weak reference to `Void`. Once again, for collectors that comprise a kind of reclaiming phase going throughout all the objects, this phase is used like the sweep phase of our mark-and-sweep and the modifications to add weak references are similar to the ones we did. However, with a collector that does not work with a reclaiming phase, such a reference-counting collector, setting the `item` field of the weak reference to `Void` can be significantly more complex.

Normally, in such a collector, an object can be reclaimed as soon as it is becomes dead, that is when its reference counter reaches zero. If this is the case, all weak references that point to this object should have their `item` attribute set to `Void`. But this would imply knowing them, which can be a problem. One way to do this could be to have a back pointer in the weakly referenced object, pointing to some list of weak reference objects pointing to it. Maintaining this list could be quite costly and burdensome, in addition to the memory overhead the list would imply, as well as the back pointer in the weakly referenced object. Using a hash table to handle these lists could avoid the use of a back pointer in each weakly referenceable object, but would still be quite expensive.

Another possibility would be to delay the actual reclaiming of the weakly referenced object when its reference counter reaches zero. In this case, the weak reference should update itself, when its `item` is queried. The algorithm for its `item` query then becomes: when `item` is already `Void`, return it; otherwise if the reference counter of the weakly referenced object is zero, set `item` to `Void` and return it; otherwise, return `item` that actually references a live object. In order to work, this also implies relying on a periodic mark-and-sweep cycle — which is anyway present in the reference counting collector because it is needed to reclaim cycles, that can not be detected by pure reference counting (see [JL96] for details). All the objects whose reference counter is zero would be actually reclaimed during this mark-and-sweep phase, and all the weak reference object updated like in our simple mark-and-sweep collector. This process is thus rather similar to what we do in our solution, except that it would imply a greater cost when accessing weak references and a delay for the reclaiming of memory.

Our solution for weak references is thus quite generalizable to other garbage collectors, although reference counting collectors make it a bit tricky and less efficient.

We explained in section 4.2.3 page 11 how the semi-conservative nature of the mark-and-sweep garbage collector in the SmartEiffel compiler influenced our

decision to avoid expanded weak references and go for normal referenced objects. Note that this makes our solution easily generalizable to all the object-oriented languages that offer referenced objects, which means probably all of them.

If the garbage collector were a non-conservative one, even for local variables, our choice for referenced weak references would still be valid. Indeed, removing the conservative scanning of the stack would simply make it possible for us to implement weak reference objects as expanded ones, but would add no extra constraint. Making the decision to go for expanded weak references would thus have to be balanced again against the arguments of sections 4.2.1 and 4.2.2 for ease of use and efficiency. Consequently, our solution for weak references does neither require a language with expanded types nor a semi-conservative garbage collector.

5 Conclusion

In this paper, we showed how a simple yet useful concept of weak references was added in a relatively easy way to the Eiffel language. We explained how this was achieved in a safe and efficient manner, thanks to the availability of parametric types in Eiffel (named generic types). We detailed how these weak references had to be implemented in the SmartEiffel compiler and the garbage collector it generates. We discussed various possible choices for weak references in Eiffel and exhibited the advantages of the solution we chose. We thus reached a mature point with a simple, elegant and efficient solution that gives the application developers much greater flexibility and control over the memory behavior of their program, when they need it, and with the minimal possible cost.

We think there are nonetheless directions for improvement in our work. We intend to go on and provide other kinds of references, besides the normal strong references and the weak references we have just described in this paper. First, we want to make it possible to have weak references that are not *all* collected when the garbage collector runs; these would be very much like our current weak references, but with a somewhat higher survival and reuse rate — if memory permits, of course. We also consider it would be a good thing to give the application developer more control over which weakly referenced objects are collected and which are not, at a given garbage collection cycle. To this end, we intend to design a kind of reference where the application developer indicates, at creation time, the "strength" of the reference, hence providing a partial ordering on weakly referenced objects. The garbage collector would then choose how many it needs to collect. We also intend to explore the possibility to give the developer full control over which weakly referenced objects are to be collected. This could be done by giving access to a lot of information on the system through introspection and reflexivity, and letting the developer use it to provide a decision about the life or dead of the weakly referenced object at run-time, when the garbage collector has to run.

References

[CCZ98] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler Support to Customize the Mark and Sweep Algorithm. In *ACM SIG-*

PLAN International Symposium on Memory Management (ISMM'98), pages 154–165, October 1998.

[CZ99] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99), Nancy, France*, pages 341–350. IEEE Computer Society, June 1999.

[Fre] Free Software Foundation. *The Guile Reference Manual*.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection*. Wiley, 1996.

[Mey92] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall Inc., 1992.

[ZC01] Olivier Zendra and Dominique Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software - Practice and Experience*, 31(6):601–613, May 2001.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141. ACM Press, Octobre 1997.